Anomaly Detectors for Self-Aware Edge and IoT Devices

Tommaso Zoppi[4]*, Giovanni Merlino[2,3], Andrea Ceccarelli[1], Antonio Puliafito[2,3], and Andrea Bondavalli[1]
[1] University of Florence, Department of Mathematics and Informatics, Viale Morgagni 65 – Florence (IT)
[2] SmartMe.io, via Salita Larderia, 98129 Messina (IT)
[3] University of Messina, Department of Engineering, 98166 Messina (IT)
[4] University of Trento, Department of Information Engineering & Computer Science, Via Sommarive 9, Povo (IT)
tommaso.zoppi@unifi.it, gmerlino@unime.it, andrea.ceccarelli@unifi.it, apuliafito@unime.it, bondavalli@unifi.it
*corresponding author

Abstract — With the growing processing power of computing systems and the increasing availability of massive datasets, machine learning algorithms have led to major breakthroughs in many different areas. This applies also to resource-constrained IoT and edge devices, which will often benefit from relatively small – but smart – local anomaly detection tasks that aim at protecting the device, or the information they convey from sensors towards a central node. This provides the device with fault detection capabilities that are typically required when engineering dependable devices, services or systems. This paper overviews a pitfall-free process to provide small devices with anomaly detection capabilities, to make them self-aware of their health condition, and possibly take appropriate countermeasures. Our methodology applies to a wide range of Linux-based devices: we show an application to a specific ARANCINO device, which has already been successfully used in many smart cities and sensing applications. We craft anomaly detectors that are very effective in detecting most of the anomalies. Additionally, we comment on the beneficial impact of time-series analysis, which could help improve detection performance even further, allowing to equip any small device with responsive and accurate anomaly detection machinery.

Keywords – anomaly detection, iot, edge, monitoring, ARANCINO

1 Introduction

Edge learning refers to the deployment of Machine Learning (ML) algorithms at the network edge [6]. The key motivation of pushing learning toward the edge is to perform on-site preprocessing and filtering of data, as well providing edge devices with sophisticated yet lightweight means to optimize their performance. Whereas the vast majority of studies on ML rely on lab setups for which we assume the availability of huge server farms, GPUs and any kind of accelerators (including FPGAs), deploying ML algorithms in the wild comes with obvious concerns [5], [6], [7]. Those are not to be intended as showstoppers but require a dedicated methodology to collect data, choose, train, test, and deploy adequate ML algorithms on devices.

It would often be desirable to bring ML algorithms in resource-constrained devices, which act as sensors, small embedded systems or controllers, or Internet-of-Things (IoT) devices in general. Those may find usage for a variety of purposes including – yet not limited to – load balancing, data preprocessing and cleaning, intrusion detection, or anomaly detection. In contrast to cloud computing, where processing and computation are centralized from a networking perspective (even though servers may be physically distributed), deploying services on the edge of the network pushes a decentralized architecture for computing, storage, and connectivity that can drastically enhance real-time applications [51]. One of the critical requirements for any device is being aware of its behavior (i.e., self-awareness); this enables the device itself to act accordingly in case of issues, and label itself as malfunctioning or performing self-diagnosis routines to identify the root cause (if any) of the behavioral anomalies in order to fix them.

Unfortunately, anomaly detection [13] is a complex data-driven task that relies on i) collecting data about the normal behavior of the target system/device, and ii) use it for training Machine Learning (ML) algorithms that can find patterns on this data that are far from expectations: those will be labeled as anomalies. From a practical standpoint, data collection, training and testing ML algorithms create a very time-consuming and resource-heavy task; thus, at the current state of the art, the problem of bringing ML to edge or resource-constrained devices is still underdeveloped and needs further research [6].

This study brings ML-based anomaly detectors to resource-constrained devices for the high-level purpose of making devices self-aware about the correctness of their behavior, or of being under an attack or intrusion. This makes devices able to monitor themselves, seek for potential performance anomalies due to errors or attacks, and activate diagnosis or recovery strategies whenever applicable. Clearly, this process has to be free from obvious pitfalls [1] that may have a detrimental impact on the entire process and will likely result in deploying a detector with doubtful usefulness. Examples include, but are not limited to: sampling bias, inaccurate threat/error model, inappropriate baseline or performance measures. We bring anomaly detectors into resource-constrained devices through 4 composite steps: i) identify the anomalies we seek to detect, ii) develop a lightweight monitor which is suitable for Linux-based devices (publicly available at [3]), iii) a)conduct experimental analyses in which the normal behavior of a device is observed, and then b) inject errors while monitoring

2693-9177/23/$31.00 ©2023 IEEE
DOI 10.1109/QRS60937.2023.00013

performance indicators, and iv) a) use the collected data to train ML algorithms for anomaly detection, b) test their behavior and c) install the most performing learned model on the device.

We apply our methodology for deploying anomaly detectors for ARANCINO [12] devices and ultimately make them self-aware about their health state. Summarizing, the novelty of this work is very relevant for the business continuity of the smart city and crowdsensing applications where devices such as the ARANCINO are being adopted. Indeed we:

- Propose a methodology for bringing anomaly detectors (ML algorithms) into resource-constrained devices that applies to all Linux-based devices.
- Show how the methodology is free from pitfalls, and that resulting anomaly detectors are able to detect most of the performance anomalies, discussing potential improvements.
- Provide source code for the monitor and the data analysis part, to increase the reproducibility of our results and provide ready-to-use means to bring our methodology over to other case studies.

The paper is structured as follows: Section 2 overviews related works and basics on IoT systems, where resource-constrained devices are most widely adopted. Section 3 overviews our target ARANCINO device, letting Section 4 expand on our methodology aimed at bringing anomaly detectors to cyber-physical devices. Section 5 discusses our experimental results, whereas Section 6 goes through potential improvements of the detector through the application of methods for time-series analysis. Section 7 concludes the paper.

2 Related Works: Bringing ML on the Edge

This section overviews related work on IoT systems and edge devices, monitoring, anomaly detection and related metrics.

2.1 IoT Systems and Edge Devices

The Internet of Things (IoT) denotes a system of interconnected physical devices and objects that utilize the Internet to communicate and exchange data. These devices can range from everyday consumer products like smart home devices to industrial tools in factories [54].

Edge devices, on the other hand, represent the "edge" of the network, acting as entry points to the core network. Edge computing allows data processing to happen close to the data source, improving response times and saving bandwidth [55].

A significant challenge in developing software for IoT and edge devices is their inherent resource limitations. These devices often have constrained computational capabilities, storage, and power supplies [56]. This imposes a restriction on the complexity of software that can be developed and deployed, prompting the need for more efficient algorithms and compact data representation.

In addition, these devices often have stringent power requirements, given their portable nature or their remote, hard-to-reach locations. Hence, the software must be optimized for low power consumption [57].

Another challenge comes from the fragmented and rapidly evolving nature of IoT platforms, where available software libraries may not be up-to-date or standardized, hindering software portability and interoperability [58].

Countering limitations related to software fragmentation, when dealing with less resource-constrained edge devices, such as those capable of hosting comprehensively capable, and standardized, operating systems, e.g., *NIX-compatible, or better still, Linux-based environments, containerization technologies (and adjacent approaches) may help, by isolating [59] the application from the hosting environment, and coupling interop efforts with container-powered deployment, i.e., CI/CD workflows' design and maintenance.

2.2 Anomaly Detection

Detecting the activation of faults is a fundamental step to achieving fault tolerance and going toward dependable systems [14]. As a result, each device must be monitored to understand if its behavior complies with expectations, or if additional actions need to be completed to assure that the system is working properly. This activity requires the deployment of *anomaly detectors*, which identify patterns that do not conform to a well-defined notion of normal behavior [13]; anomalies may be the symptom of faults, attacks, or upcoming failures. Recent trends [27], [40], [41] show how data-driven detection may provide effective and flexible means to detect anomalies, as opposed to traditional rule- and signature-based mechanisms. Data-driven detectors are usually implemented as binary classifiers (simply called *classifiers* in the rest of the paper), which are Machine Learning (ML) algorithms that assign either a positive or a negative class to each data point. The negative class is mapped to the normal class (no anomaly), whereas the positive class is known as the anomaly class. Ideally, an anomaly detector will be able to correctly classify the state of the device, minimizing misclassifications – either false alarms (False Positives, FP) or missed detection of anomalies (False Negatives, FN).

Literature and practice tell us that the vast majority of anomaly detectors for tabular data are implemented through supervised classifiers [25], [34]. They require training data for which the label is known and build a model that is usually accurate, i.e., only a few misclassifications occur. Well-known families of supervised classifiers are: i) tree-based, mostly decision trees, ii) statistical techniques [26], iii) distance-based learners [25], iv) support vector machines [24], and deep neural networks (DNNs) [27], [30], [31]. DNNs are classifiers structured with many hidden layers and are the de facto standard for classifying unstructured data such as images, audio, lidar point clouds, and videos; however, they often struggle when classifying tabular data. For instance, recent works [27], [29], [30], [31], [42] advocate that deep neural network classifiers applied to tabular data have worse classification performance than other

supervised classifiers. There are also studies that convert tabular data into images to fully exploit the potential of deep learners in processing images [28], but classification performance does not benefit much. On top of that, DNNs are heavy models that do not pair well with devices that have limited processing and storage resources, as is common for IoT devices.

Anomaly detectors can also be implemented using unsupervised classifiers, which do not require labels in the training data: they build their model under the assumption that anomalies due to hw/sw faults or ongoing attacks manifest as observable deviations from the nominal behavior. This makes unsupervised classifiers applicable even when a labeled training set is not available; on the downside, they usually generate a higher number of misclassifications – especially false positives – than supervised classifiers [32], [33], and are beneficial only when unknown anomalies or zero-day attacks are a concern [42].

2.3 Monitoring IoT Devices and Embedded Systems

Anomaly detectors need to learn how the system behaves normally: the enabling knowledge to complete this process are the features that are continuously monitored from the target device. Features are usually obtained by monitoring the performance indicators of a device at the hardware or low level [36], system level [9], [39], input/sensor [35], environment [37], application level (e.g., SCADA [38]) or even coding level [30]. Features can be textual or numeric: textual features (e.g., the name of a protocol) are always categorical, while numeric features may either be categorical (e.g., the ID of a system call) or describing an ordinal range of values, e.g., the percentage of memory used, or the number of packets received from the network interface in a time frame. Categorical features usually require preprocessing before being fed to a classifier. In fact, classifiers may compute algebraic calculus such as Euclidean distance [25] or the estimation of angles in a multidimensional space, which delivers misleading results when processing categorical features.

2.4 Metrics to Evaluate Anomaly Detectors

The classification performance of anomaly detectors is typically expressed using metrics [44] that compute correct classifications, True Positives (TPs) and True Negatives (TNs), and misclassifications, False-Positives (FPs) and False Negatives (FNs), respectively. These metrics are usually referred to as acan be aggregated into a wide variety of compound metrics, among others: False Positive Rate, Precision, Recall (or Coverage), and many others. In this paper, we will mostly use the following ones, which account for all 4 items of the confusion matrix and as such provide a balanced view on FPs and FNs..

Accuracy (ACC) calculates the percentage of correct classifications (TPs and TNs) over all classifications. Importantly, 1 - ACC is usually referred to as the misclassification rate.

$$ACC = \frac{TP + TN}{TP + TN + FP + FN}$$

The *Matthews correlation coefficient* (MCC) [43] is particularly suited when the dataset is unbalanced [2], i.e., the occurrences of normal data points and anomalous data points are significantly different. This situation occurs quite frequently in the context of embedded systems or smart sensors, which are expected to behave as expected in most cases, observing only a few exceptions over time.

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

3 ARANCINOs: Enabling ML on the Edge

ARANCINO™ [12] is the trade name for a family of IoT and embedded boards based on its namesake architecture. SmartMe.io (https://smartme.io), a startup company from Messina (southern Italy), conceived this architecture from the ground-up, off a composite mix of state-of-the-art open source (and open hardware) building blocks, and is routinely crafting Industry 4.0 and Smart City solutions, for its customers around the world, based on ARANCINO-compliant boards exclusively. ARANCINO-enabled applications [53], [59] in the wild so far include, among others: weather/pollution monitoring, smart city public lighting, trailers' mobility/health monitoring, bridges'/roads' structural safety.

3.1 Building blocks of an ARANCINO

As depicted in Figure 1, the hardware/software architecture embodied by the ARANCINO blueprint may be exemplified by a (custom-built) Single-Board Computer (SBC)-compatible layout, which features:

- a Carrier Board, which runs a MicroProcessor Unit (MPU), from Raspberry Pi-family Compute Modules (RPi-CMs), which is a cheap, credit card-sized ARM64-based GHz-class Linux-compatible computers;
- one (or more) on-board MicroController Units (MCUs), such as Microchip SAMD21-family ARM Cortex-M0+-
- (on-module) RAM (e.g., 1GB) and (flash) storage (e.g., 8/16/32GB eMMC)

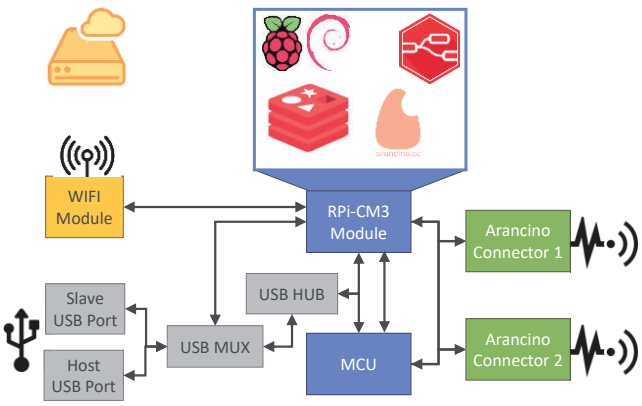

Figure 1. An high-level overview of the ARANCINO hardware/software architecture

- two (or more) standard (i.e., mikroBUS™ Click) ARANCINO Connectors, for peripherals (e.g., WiFi/BT interfaces), transducers and all-round expandability;
- other slots, most notably: 1 SPI, 25 GPIOs (pin sharing configuration) 3 USB Host + 1 USB device (used to program Compute Modules);
- A crypto-chip can be made available at will for hardware-based secure key storage.

Those blocks are physically mapped onto the device as showed in Figure 2.

3.2 Software running on the MPU

In terms of the (regular) software stack on the ARANCINOs, there are four (core) components, as discussed below.

The MPU (RPi-CM3) runs a Linux-based fork of Raspbian OS, adapted to ARANCINO, which hosts

- the lightweight in-memory key-value (KV) *Redis* storage;
- the *arancino-daemon* service, which manages communication over UART (USB) with the built-in MCU, as well as with (additional) units when plugged-in over USB; recent version support communication over Bluetooth, and publisher/subscriber messaging as well;
- the *lightning-rod* (LR) low-level (*watchdog*-like) service, to i) integrate the board into the Cloud infrastructure, ii) forward timeseries data, and iii) support plugins used to interact with the Cloud dashboard.

No special drivers are required. Driver-enabled I/O may be then remotely accessed through the (I/O)Cloud [72], and attached to remote Cloud-hosted IoT boards and/or VMs transparently. The Cloud stack as well is built around open-source technologies and extends the widely adopted OpenStack ecosystem with a custom subsystem devoted to IoT nodes onboarding, presence, and runtime management/customization, called *IoTronic*, part of the umbrella Stack4Things [72].

3.3 Software running on the MCU and Interconnections

On the MCU, an ARANCINO runs the (application-specific) C/C++ firmware in compliance with the *arancino-library*, which provides suitable primitives to read data off

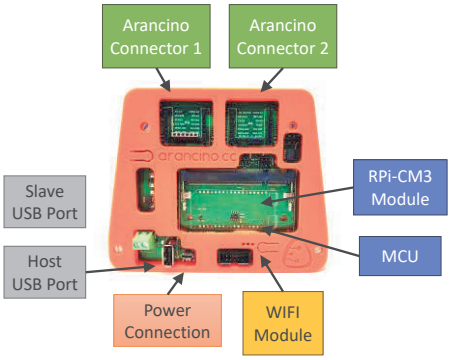

Figure 2. An instance of a board in the ARANCINO family, depicting its layout and mapping it to the main components as identified in Figure 1

(and store data to) the MPU-side business logic leveraging the Redis-backed KV store.

The arancino-daemon on the MCU side and arancino-library on the MPU side implement *Cortex*, a "no-frills" ARANCINO-oriented communication protocol. Its name intentionally refers to the cerebral cortex in the human brain, where the cortex acts as a conduit between right and left hemispheres of the brain. This split separates the real-time processing that typically interacts with transducers and often required to act promptly on the MCU from more elaborate computing duties (e.g., ML-based inference at the edge, data sync/transfer to the Cloud), which are almost batch-like on the MPU. More in general, the "split-brain" approach leaves room for a self-aware, *proprioception*-like approach where part of the system takes care of introspective monitoring duties, and the other part infers the health status of the node overall.

4 A Methodology to Design Self-Checking Devices

Our methodology to deploy anomaly detectors for resource-constrained IoT and edge devices relies on the following 4 steps.

S1. Create an error model that covers most of the anomalies that can be generated by the manifestation of common errors in Linux-based IoT devices.
S2. Create a monitoring system that fits our case study but also applies to similar devices.
S3. Perform error injection campaigns in which we monitor the behavior of the target device under normal operating conditions and when errors are injected.
S4. Use collected data to train anomaly detectors that can then be deployed in the target device to monitor their detection and timing performance.

Each step is detailed in Sections 4.2 to 4.5.

4.1 Dealing with Common Pitfalls

Before getting into details, we want to make sure that our methodology is free from obvious pitfalls, or to adequately mitigate the issues that are likely to arise in our process. To accomplish that, we refer to the paper [1], which lists 10 pitfalls that are likely to impact studies as this one. We report each pitfall in Table I, discussing if and how likely it may be to impact any of our steps S1 to S4, and mitigations to avoid pitfalls posing threats to the validity of our study. Notably, the pitfalls are mostly related to the data analysis and machine learning processes, which happen in S4. However, the successful mitigation of those pitfalls is due to actions that need to be taken in earlier steps (S1 – S3) as well. Therefore, these pitfalls are cross-cutting to our process and thus highly representative.

4.2 S1 – An Error Model for Linux-Based Devices

Detecting anomalies assumes the knowledge and characterization of how the system behaves normally [13], but benefits also from information about the effects of potential faults or attacks (i.e., errors [14]) that may impact

Table I. Pitfalls for machine learning in computer security, from [1], mapping to the steps of our methodology, and mitigations to be applied to avoid the occurrence of each pitfall in our study.

Pitfall (from [1])	S1	S2	S3	S4	Mitigation
P1 – Sampling Bias. The collected data does not sufficiently represent the true data distribution of the underlying problem	✓	✓	✓		The error model should cover a wide range of performance anomalies that may happen in the target device. Also, the monitoring system should gather information from different areas of the system to have a complete view of the problem
P2 – Label Inaccuracy. The ground-truth labels required for classification tasks are inaccurate, unstable, or erroneous, affecting the overall performance of a learning-based system			✓		Our system monitoring campaign should annotate the timeframe in which each error is injected into the system, allowing precise and flawless labeling.
P3 – Data Snooping. A learning model is trained with data that is typically not available in practice.	n.r.	n.r.	n.r.	n.r.	Our monitoring campaign is exercised on a real device that is currently used in many scenarios: therefore, this pitfall is irrelevant for our study.
P4 – Spurious Correlations. Artifacts unrelated to the problem create shortcut patterns for separating classes.		✓		✓	Our monitoring system should avoid gathering features that contain information about the experimental setup rather than information on the detection problem itself.
P5 – Biased Parameter Selection. The final parameters of a learning-based method are not entirely fixed at training time. Instead, they indirectly depend on the test set.				✓	The test set is clearly separated from the training set, with no overlaps.
P6 – Inappropriate Baseline. The evaluation is conducted without, or with limited, baseline methods.				✓	Albeit a complete benchmark is not the main aim of the paper, we will compare the detection and timing performance of 9 supervised anomaly detectors.
P7 – Inappropriate Performance Measures. The chosen performance measures do not account for the constraints of the application scenario, such as imbalanced data or the need to keep a low false-positive rate				✓	We are aware that common metrics such as accuracy or F1-Score should be disregarded [2] when evaluating binary classifiers that detect performance anomalies in IoT devices. Those are expected to perform regularly, and only rarely encounter problems (thus the real-world data they face is skewed and unbalanced towards normal data).
P8 – Base Rate Fallacy. A large class imbalance is ignored when interpreting the performance measures leading to an overestimation of performance.				✓	
P9 – Lab-Only Evaluation. A learning-based system is solely evaluated in a laboratory setting, without discussing its practical limitations.			✓		This paper conducts a lab-only evaluation. However, we try mitigating this problem by conducting system monitoring under different operating conditions.
P10 – Inappropriate Threat/Error Model. The security of machine learning is not considered, exposing the system to a variety of attacks, such as poisoning and evasion attacks.	✓			✓	This study does not account for adversarial attacks to the anomaly detector. However, we make sure that our error model is as much correct and complete as possible.

the target system. Therefore, we contacted the stakeholder to discuss how the target ARANCINO device was made, potential vulnerabilities, the existence of bottlenecks, and relevant software or communication channels. Then, we scanned the literature to seek for error models that apply to a Linux-based embedded system / IoT device [8], [9], [10]. There is an overall agreement about the high likelihood of one of the following events happening in a Linux-based OS.

- Resource consumption: either CPU, primary and secondary memory may be filled/exhausted by malicious or malfunctioning software.
- Deadlock: critical sections are heavily used in any multi-threading context. Shallow management of locks or semaphores may end up generating deadlocks and make the regular execution flow deviate from expectations.
- Unexpected usage of the network, in both directions.

On top of that, we consider that ARANCINO devices heavily rely on the Redis [4] database: therefore, we also consider erroneous usages of the Redis database, which we simulate as subsequent reads / write operations. Lastly, we disturb the regular usage of key processes that manage the overall device, namely the *arancino* and *node-red* Raspbian processes, and make them stuck for some time to simulate their potential malfunction.

This leads to a total of 8 different errors (CPU usage, RAM usage, Disk Usage, Deadlock, Redis read, Redis write, Stuck arancino, Stuck node-red) that we will inject into our device, monitoring its behavior in the process. We believe that this set of errors is rather extensive and refers to different modules, processes, and software components. Obviously, it may not be complete, but we believe it is good enough to successfully address and mitigate pitfall P1 and, to a lesser extent, P10.

4.3 S2 – A Lightweight Monitor for Linux-based Devices

We first looked for existing software with the following requirements:

- lightweight;
- customizable regarding sampling interval and the system indicators to observe;
- able to instrument different layers and components of the target system;
- compatible with the *Raspbian 9 Stretch* system, the OS running on the ARANCINO devices. This means that the

tool must be written either in C/C++ (gcc 7.x), Python <= 3.5.3, or Java (v. 8 openJDK).

Unfortunately, we did not find any good fit for that: as such, we coded a monitor ourselves, and made it publicly available through a public GitHub repository [3]. The monitor is written in Python 3.5.3, featuring a total of 7 probes, that can be activated at will:

- Network (32 features): reads data from the system file */proc/net/dev*
- Chip temperature (1 feature): reads data from the system file */sys/class/thermal/thermal_zone0/temp*
- Virtual Memory (116 features): reads data from the system file */proc/vmstat*
- Memory Info (38 features): reads data from the system file */proc/meminfo*
- IO Stats (6 features): uses the *iostat* Linux package and parses its textual output.
- Python Indicators (55 features): uses the *psutil* functions *cpu_times, cpu_stats, getloadavg, swap_memory, virtual_memory, disk_usage, disk_io_counters, net_io_counters*.
- Redis DB (25 features): accesses to Redis performance indicators through the *redis-py* Python wrapper

The reader should note that this monitor has only minimal dependencies and thus can be installed without requiring to download additional libraries. For further information, please refer to the documentation available at [3]. Also, it activates a single process in the target device, making it easy to control for *intrusiveness*, which is a major concern when setting up this type of study. Ideally, a monitor should observe the performance indicators of a system without impacting it [15]: however, this is possible only in restricted scenarios e.g., network monitoring using an external device connected to the network. Conversely, we need to deploy the monitoring software to the target device, partially using the resources that we are going to monitor with the software itself. Therefore, our monitor will unavoidably gather information that is affected by the fact that the monitor itself is running and using the device's resources. However, this does not represent a problem nor creates spurious correlations (pitfall P4) for the following two reasons. Firstly, the activity of the monitor is constant over time: therefore, the added load is semi-constant such that it can be considered background noise. Secondly, the monitor is always active in our experiments: therefore, it will affect all data we gather in the same way and will not make any difference with respect to the boundaries between normal and anomalous behavior.

4.4 S3 - Injecting Errors to Collect Labeled Data

The definition of the error model and the monitoring system paves the way for an experimental campaign in which we equip the ARANCINO with the monitoring system that samples data every second, and let it do its business as usual. This provides a data baseline that we can use to characterize the normal behavior of the device. Additionally, we perform random injections of each of the 8 errors in Section 4.2, keeping the monitoring system in place. This provides data about how the ARANCINO device reacts to errors (i.e., the anomalies due to different root causes), which we label accordingly. Our experimental setup activates an injection with 5% of probability. The error to inject is randomly chosen out of the 8 available in our error model. Once activated, the injection remains active for 5 seconds; then, we manually deallocate as best as we can the resources used for the injection and wait a total of 10 seconds of cooldown. During cooldown it is not possible to activate new injections, providing the device with some time to repair itself (e.g., garbage collection, freeing up zombie processes, de-allocating resources that were used by the injection and that could not be manually freed, and so on).

This methodology provides a very precise and reliable way of gathering labeled data without any inconsistencies (thus avoiding pitfall P2). On the other hand, the reader may argue about the representativeness of the data we collect with respect to the behavior of a device that is meant to be used in different setups as an edge component of an IoT system, or as a smart sensor. Conducting experiments in which the ARANCINO is performing the same task in the same network topology, with the same operating conditions and workload may heavily bias the data we have and thus skew the detecting phase (pitfall P9). There are no means to avoid this problem at all: we mitigate it by exercising experiments in the following 3 environmental setups:

- *uni-env* (69000 observations): the device is connected to a WiFi network at the premises of our university; the network is shared by many devices and features a non-trivial topology.
- *home-env* (72000 observations): the device is connected to a WiFi network in a flat of a residential building, where only a few devices are admitted to access the network. It is directly connected to the router (no intermediate switches) and to the FTTC private fiber network.
- *out-env* (13000 observations): the device is positioned outdoors and connected to a private WiFi network (i.e., tethering from a mobile connection) providing connectivity to two nodes at all times.

This leads to the creation of 4 separate labeled datasets [52] composed of the timestamp, i.e., a *long int* in ms, 276 features from the monitoring system, and a label i.e., normal or any of the 8 errors: a dataset for each scenario above, and a fourth dataset which merges the first 3 datasets (*all-env,* composed of 154000 observations).

4.5 S4 - Anomaly Detectors

A labelled dataset enables the usage of any supervised ML algorithm for detecting performance anomalies. This opens doors to a plethora of different experiments and comparisons about the detection performance among a multitude of algorithms. Particularly, literature tells that the de-facto standard approach for processing tabular datasets (as ours actually is) consists in employing tree-based ML algorithms,

which typically outperform neural networks, even those that are being repurposed explicitly to classify tabular data [11]. Additionally, we are interested in selecting a subset of classifiers that are as heterogeneous as possible to avoid exercising many classifiers which will result in very similar outcomes (see pitfall P6). We favor classifiers that require minimal parameter tuning to avoid conducting random or grid searches which would add yet another dimension of analysis and expose to pitfall P5.

Therefore, we selected the statistical Gaussian, Bernoulli and Multinomial Naïve Bayes [20], a Perceptron, the Logistic Regression [18], as well as the tree-based Decision Tree [23], Random Forest [22] and Gradient Boosting [17], whose implementations are all made available in the Scikit-Learn Python package. Remember that we are constrained to using Python 3.5.3, which locks the Scikit-Learn version to 0.22.1 (3 years old and missing recent algorithms and features). Also, we do not consider classifiers that have an $O(n^2)$ usage of memory, since they make the training process fail due to insufficient memory. Thus, algorithms such as Linear [19] and Quadratic [16] Discriminant Analysis, Support Vector Machines [18] and any neural network other than the single-layer perceptron are out of the picture. Also, we avoid the usage of K-th Nearest Neighbors [21], which even with the kd-tree enhanced neighbor search requires an exceedingly high $O(\log n)$ amount of time to decide on anomalies and thus cannot be applied as an anomaly detector for the ARANCINO. Overall, we select 9 algorithms that perform well with default parameter values: however, for ensemble classifiers as random forests and gradient boosting, where the number of estimators has a huge impact on resource usage, we create instances using 10, 30 and 100 estimators. This leads the total of supervised algorithms to 13: gaussian naïve bayes (gnb), Bernoulli naïve bayes (bnb), multinomial naïve bayes (mnb), perceptron (mlp), logistic regression (lr), decision tree (dt), stochastic gradient descend (sgd), random forest with 10, 30 and 100 trees (rf10, rf30, rf100), gradient boosting with 10, 30, and 100 decision stumps (gb10, gb30, gb100).

We want ML algorithms to act as anomaly detectors: therefore, we will convert the 9-class label of the datasets in S3 to a binary label, i.e., normal against others. Also, we preprocess datasets to remove hidden labels and constant columns that represent useless features. This leads to a set of 119 (out of the initial 276) non-trivial features to be used for detection in each dataset. Once everything is set, we run each of the 13 supervised classifiers on each of the 4 datasets (for a total of 52 models), and test against all 4 datasets as well. Noticeably, there is a general agreement that ML algorithms should not be necessarily trained on edge or in any other device that has strong resource constraints, whenever possible. In this situation, the model is trained somewhere else, and the resulting model is then sent to the target device, deployed and ready to use at runtime. However, there may be situations in which this is not feasible due to reasons such as data privacy, or specific system architectures (e.g., a federated learning scenario [45]). Therefore, we will train and test anomaly detectors on the device itself.

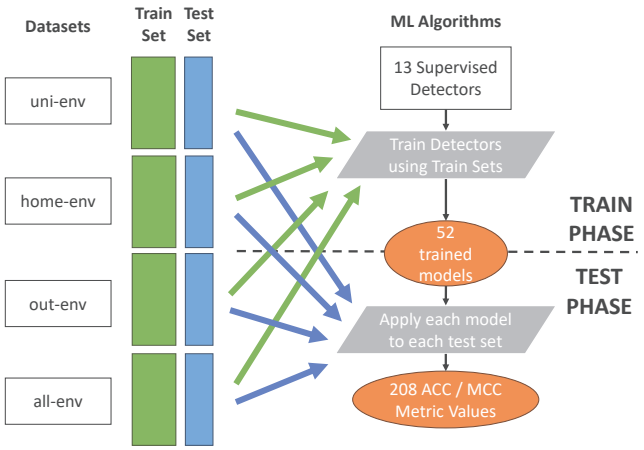

Figure 3. The process of training and testing anomaly detectors in this paper.

We measure the detection performance of each model through the confusion matrix (i.e., TP, TN, FP, FN), Precision (P), Recall (R, or coverage), False Positive Rate (FPR), Accuracy (ACC) and Matthews Correlation Coefficient (MCC). MCC is robust to unbalanced datasets and as such nice to pair with accuracy to address pitfalls P7 and P8. To guarantee independence between train and test sets, we proceed to a 70-30 train-test split of each dataset. The summary of this process is in Figure 3.

5 Experimental Results and Discussion

This section shows the experimental results related to the training (Section 5.1) and testing phase (Section 5.2) of anomaly detectors on the target ARANCINO device.

5.1 Training Anomaly Detectors

We first analyze the training phase of anomaly detectors. Figure 4 plots train time (in seconds) against model size (in KBs) of each of the 13 anomaly detectors trained using all-env, out-env, uni-env datasets. Results using home-env do

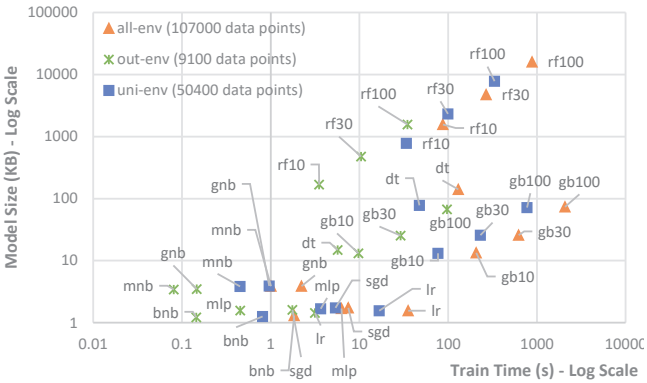

Figure 4. Plotting train time (seconds) against model size (in KB) of each anomaly detector trained using each of the 4 datasets. Results using home-env do not appear here as they overlap with those of uni-env. Logarithmic scale for both axes.

not appear in the figure as they overlap with those of uni-env. Let us focus on the horizontal axis: the more we go to the right, the slower the training process. As expected, most of the items on the right of the plot correspond to detectors trained using all-env and uni-env datasets, which contain many more data points than out-env. The difference may not seem relevant, but the reader should consider that the plot has logarithmic scale on both axes. There is another important trend: gradient boosting and random forest detectors are always more on the right with respect to other detectors trained using the same dataset, whereas the naïve bayes detectors are always faster and hover on the left of the picture. The vertical axis of Figure 4 instead depicts the size of the models at the end of the training phase, which follow a similar trend with respect to the train time. Overall, it is safe to say that the detectors that require more time to train are also the ones that output a heavier model. This may not seem surprising, but it is not a trivial observation: we can observe how logistic regression *lr* does not exactly follow this trend, as it produces a few KB of model, but requires far more time than fast detectors such as the naïve bayes *gnb, bnb, mnb*.

5.2 Testing Anomaly Detectors

Another important discussion regards the performance of anomaly detectors both from a classification and timing viewpoint. We expand on this item with the aid of Figure 5, which plots test time (in milliseconds) against MCC of each anomaly detector tested against home-env dataset. Depicting results when testing all detectors on all datasets would have made the plot completely unreadable, thus we selected a single test dataset. Other results are in the repository at [52]. The scatterplot may still seem hardly readable due to the high count of items and labels, yet it provides the following key insights.

- All detectors can classify a single data point in at most 167 ns (0.167 ms), which is very fast and thus does not represent a timing bottleneck when deploying these algorithms on the ARANCINO.
- Up in the picture (i.e., high MCC and thus good detection performance) we exclusively find yellow circles: there is no detector trained using *all-env, out-env, uni-env* that has an MCC higher than 0.6 when tested against home-env. High MCCs are achieved by detectors trained and tested on different partitions of the same home-env dataset, meaning that each of the 4 dataset is slightly different from others, and that models do not always generalize well when train and test sets come from different datasets.
- Overall, *rf* and *gb* classifiers have a better MCC than other detectors when dealing with the same dataset. Also, only rf and gb detectors reach and surpass MCC = 0.5, a result that shows there is still room for improvement.

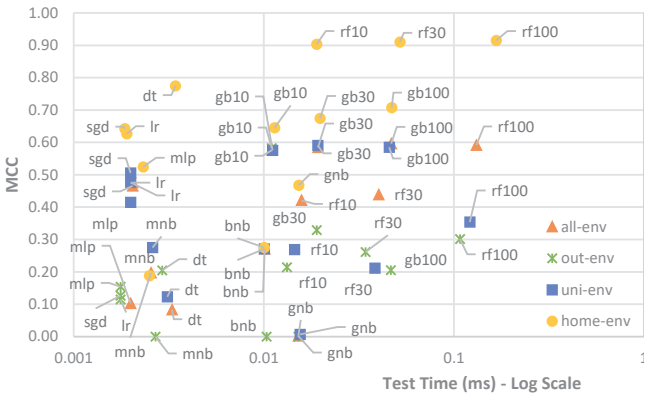

Figure 5. Plotting test time (milliseconds) against MCC of anomaly detectors trained using home-env(yellow dots), uni-env (blue squares), out-env (green marks), all-env (orange triangles) and tested against the same home-env dataset. Logarithmic scale for horizontal axis.

The reader may have expected that using all-env as training set would have provided a more complete and robust

Table II. Timing and Classification performance of bnb, lr, gb10, rf100 detectors trained on different datasets and tested against home-env. Classification metrics are paired with arrows that specify if we want the metric to be high (↑) or low (↓).

	clf	Train dataset	time (ns) (↓)	ACC (↑)	MCC (↑)	TP (↑)	TN (↑)	FP (↓)	FN (↓)	FPR (↓)	P (↑)	R (↑)
Statistical	bnb	all-env	6.1	0.844	0.271	387	17850	3309	54	0.156	0.105	0.878
		home-env	6.0	0.845	0.275	384	17862	3312	42	0.156	0.104	0.901
		out-env	6.2	0.171	0.000	3696	0	0	17904	n.a.	1.000	0.171
		uni-env	6.0	0.844	0.271	384	17853	3312	51	0.156	0.104	0.883
	lr	all-env	2.0	0.874	0.476	1037	17844	2659	60	0.130	0.281	0.945
		home-env	1.9	0.905	0.627	1864	17680	1832	224	0.094	0.504	0.893
		out-env	1.8	0.397	0.114	3066	5504	630	12400	0.103	0.830	0.198
		uni-env	2.0	0.864	0.477	1786	16878	1910	1026	0.102	0.483	0.635
Tree-based Ensembles	gb10	all-env	11.0	0.893	0.576	1391	17899	2305	5	0.114	0.376	0.996
		home-env	11.4	0.908	0.645	1746	17873	1950	31	0.098	0.472	0.983
		out-env	11.0	0.895	0.582	1674	17664	2022	240	0.103	0.453	0.875
		uni-env	11.1	0.893	0.575	1391	17896	2305	8	0.114	0.376	0.994
	rf100	all-env	131.2	0.897	0.591	1627	17752	2069	152	0.104	0.440	0.915
		home-env	167.0	0.976	0.915	3226	17864	470	40	0.026	0.873	0.988
		out-env	107.4	0.704	0.301	2485	12717	1211	5187	0.087	0.672	0.324
		uni-env	121.1	0.786	0.354	2107	14863	1589	3041	0.097	0.570	0.409

model that consequently would have had the highest metric scores on any test set. However, the all-env dataset includes data from different experimental setups that (slightly) differ from each other and may represent an heterogeneous baseline that constitutes noise instead of helping to build a more robust model. In our case, training using all-env is not detrimental, but also not really beneficial when looking at metric scores.

For completeness, Table II reports the classification metrics related to some of the detectors in Figure 5. We show all classification metrics and the time to predict the label of a data point related to statistical anomaly detectors *bnb* and *lr*, and to ensembles of decision trees *gb10* and *rf100*. There are 4 rows for each anomaly detector, reporting scores related to each of the 4 train datasets tested against the same home-env. Starting from the time(ns) column, we can observe how the statistical detectors are clearly faster than tree-based *gb* and *rf*. The time needed by *rf100* is far higher than its competitors: this is not strictly due to the algorithm, but to the number of trees (100) in the forest. The reader can notice that the time needed by *gb10* is clearly faster, but this is because the *gb10* detector is an ensemble of 10 weak learners, a tenth of the trees in the *rf100*.

We then switch our focus to classification metrics, with Accuracy (ACC) being the most used typically. We see that ACC scores of the statistical detectors are inferior with respect to those of tree-based ensembles, which is expected as the latter are conceived to output a low number of misclassifications. The trend holds when looking at different metrics, which keep showing better results overall when using tree ensembles. We want to point out how the same algorithm trained with different training sets has very different detection performance when tested against the same test set: this is particularly evident for *bnb*, *lr* and *rf100*, which have very poor detection performance when trained using the out-env dataset. This is due to the training dataset not being informative enough or due to the algorithm not learning a model which generalizes well to datasets collected in similar (yet not identical) operating conditions. Instead, the classification performance of *gb10* is very similar across the 4 test datasets, because it builds a robust model that does not suffer from the problem above mentioned.

6 On Time Series Analysis

However, Table C raises an important discussion item: even gb10 - which has the most consistent classification performance out of all the detectors in this paper – still outputs several misclassifications which may be considered high in most cases. Particularly, an accuracy of roughly 90% means that one observation out of 10 will be misclassified, most likely as a false positive (i.e., low precision, very high recall). This generates several false alarms which may be considered unfeasible for practically deploying anomaly detectors in the ARANCINO device as it will trigger too many unnecessary investigations to diagnose a problem that in fact does not exist.

There seem to be straightforward ways to deal with this problem: just generate a bigger training dataset or try more algorithms and more configuration of hyperparameters to find the sweet spot that maximizes the classification performance. However, the detectors we trained up to this moment are not exploiting the fact that the behavior of the target device is meant to (gracefully) evolve through time. Instead, we are labeling data points in the test set solely relying on each of them separately, instead of making a prediction based on the current observation plus the way feature values evolved in the (recent) past. In other words, we clearly want to classify data points in time series, but we are neglecting this property ourselves by using ML algorithms that perform classification without the notion of time ordering.

For the sake of brevity, we cannot discuss the time-series approach within the size of this paper; instead, we list below the approaches we are currently planning to and currently using to enhance classification performance of the anomaly detectors.

- Using ML algorithms that are naturally meant to process time series as anomaly detectors. This approach is straightforward, but it carries an important problem: algorithms such as LSTM [46] employ convolutional neural networks, which are typically considered some of the heaviest classifier to train and test, and as such do not pair well with resource-constrained environments such as small embedded devices or edge devices.
- Hand-crafting new features that carry time-series information. For example, if an existing feature is “percentage of RAM used”, we may be interested in deriving features as “difference in percentage of RAM used with respect to the previous observation”, or similar others. This does not put constraints on the ML algorithms to use but creates anomaly detectors that rely on many more features and as such may be slower to exercise.
- Exploring specific approaches, e.g., bag-of-features [47], complex temporal features [48], feature fusion [49], Gaussian Process Regression [60] and, more importantly, distance-based [50] features, which may be global, local or embedded depending on the way they are computed.

7 Conclusions

This paper presented a methodology for bringing Machine Learning (ML) into resource-constrained devices to provide them with means to detect behavioral anomalies and thus be self-aware of their health status. We presented a general methodology that may be applied to any edge, embedded or IoT device which runs a Linux-based OS (indeed, the vast majority on the market), describing common pitfalls and how our methodology is robust with regard to these. Then, we showed an application of the methodology targeting ARANCINO devices, which are equipped with sensors, an MCU, and a Raspbian-derived OS that puts everything

altogether and provides connection primitives and basic OS services. This allowed ARANCINOs to find wide application in different domains such as smart cities, environmental and transportation monitoring: however, enabling them to auto-detect behavioral anomalies may actually make these devices genuinely unique on the market.

Applying our methodology helped us collect datasets about the behavior of the ARANCINO device, injecting performance anomalies and observing how the device reacts. Those datasets were used to train and test ML algorithms suitable for binary classification and thus excellent at detecting anomalies. However, those algorithms do not consider data as time-ordered series, and as such cannot precisely define the evolution of the context. Therefore, as a future work, we are and will be exploring and experimenting with time-series analysis, which we expect to have a beneficial impact on the whole classification task, particularly in lowering false alarms and improving our overall detection performance.

ACKNOWLEDGMENTS

This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.
We also thank Maurizio Giacobbe, Nicola Peditto and Fabio Verboso from SmartME for the fruitful discussions and technical support.

REFERENCES

[1] Arp, D., Quiring, E., Pendlebury, F., Warnecke, A., Pierazzi, F., Wressnegger, C., ... & Rieck, K. (2022). Dos and don'ts of machine learning in computer security. In 31st USENIX Security Symposium (USENIX Security 22) (pp. 3971-3988).

[2] Chicco, D., & Jurman, G. (2020). The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. BMC genomics, 21, 1-13.

[3] Monitor GitHub (online), https://github.com/tommyippoz/arancino-monitor

[4] Carlson, J. (2013). Redis in action. Simon and Schuster.

[5] Murshed, M. S., Murphy, C., Hou, D., Khan, N., Ananthanarayanan, G., & Hussain, F. (2021). Machine learning at the network edge: A survey. ACM Computing Surveys (CSUR), 54(8), 1-37

[6] Zhu, G., et. al. (2020). Toward an intelligent edge: Wireless communication meets machine learning. IEEE communications magazine, 58(1), 19-25.

[7] Merenda M, Porcaro C, Iero D. (2020). Edge machine learning for ai-enabled iot devices: A review. Sensors, 20(9), 2533

[8] Koopman, P., Sung, J., Dingman, C., Siewiorek, D., & Marz, T. (1997, October). Comparing operating systems using robustness benchmarks. In Proceedings of SRDS'97: 16th Symp. on Reliable Distributed Systems (pp. 72-79). IEEE.

[9] Zoppi, T., Ceccarelli, A., Bondavalli, A. (2019). MADneSs: A multi-layer anomaly detection framework for complex dynamic systems. IEEE Transactions on Dependable and Secure computing, 18(2), 796-809.

[10] Chou, A., Yang, J., Chelf, B., Hallem, S., & Engler, D. (2001, October). An empirical study of operating systems errors. Proc of the 18th ACM Symp. on OS principles (pp. 73-88).

[11] Gorishniy, Y., et. al. (2021). Revisiting deep learning models for tabular data. Advances in Neural Information Processing Systems, 34, 18932-18943.

[12] Giacobbe, M., Alessi, F., Zaia, A., & Puliafito, A. (2020). Arancino.cc™: an open hardware platform for urban regeneration. International Journal of Simulation and Process Modelling, 15(4), 343-357

[13] Chandola, V., Banerjee, A., & Kumar, V. (2009). Anomaly detection: A survey. ACM computing surveys (CSUR), 41(3), 1-58

[14] Avizienis, A., Laprie, J. C., Randell, B., & Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. IEEE transactions on dependable and secure computing, 1(1), 11-33.

[15] Spanoudakis, G., & Mahbub, K. (2006). Non-intrusive monitoring of service-based systems. International Journal of Cooperative Information Systems, 15(03), 325-358.

[16] Srivastava, S., Gupta, M. R., & Frigyik, B. A. (2007). Bayesian quadratic discriminant analysis. Journal of Machine Learning Research, 8(Jun), 1277-1305.

[17] Chen, T., & Guestrin, C. (2016, August). Xgboost: A scalable tree boosting system. In Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining (pp. 785-794).

[18] Robles-Velasco, A., Cortés, P., Muñuzuri, J., & Onieva, L. (2020). Prediction of pipe failures in water supply networks using logistic regression and support vector classification. Reliability Engineering & System Safety, 196, 106754

[19] Fisher, R. (1936). Linear discriminant analysis. Ann. Eugenics, 7, 179.

[20] Rish, I. (2001, August). An empirical study of the naive Bayes classifier. In IJCAI 2001 workshop on empirical methods in artificial intelligence (Vol. 3, No. 22, pp. 41-46).

[21] Liao, Y., & Vemuri, V. R. (2002). Use of k-nearest neighbor classifier for intrusion detection. Computers & security, 21(5), 439-448.

[22] Breiman, Leo. "Random forests." Machine learning 45 (2001): 5-32.

[23] Patel, H. H., & Prajapati, P. (2018). Study and analysis of decision tree based classification algorithms. International Journal of Computer Sciences and Engineering, 6(10), 74-78.

[24] Hearst, Marti A., et al. "Support vector machines." IEEE Intelligent Systems and their Applications 13.4 (1998): 18-28.

[25] Liao, Yihua, and V. Rao Vemuri. "Use of k-nearest neighbor classifier for intrusion detection." Computers & Security 21.5 (2002): 439-448.

[26] Srivastava, S., Gupta, M. R., & Frigyik, B. A. (2007). Bayesian quadratic discriminant analysis. Journal of Machine Learning Research, 8(Jun), 1277-1305.

[27] Zhang, C., Jia, D., Wang, L., Wang, W., Liu, F., & Yang, A. (2022). Comparative Research on Network Intrusion Detection Methods Based on Machine Learning. Computers & Security, 102861.

[28] Zhu, Y., Brettin, T., Xia, F., Partin, A., Shukla, M., Yoo, H., & Stevens, R. L. (2021). Converting tabular data into images for deep learning with convolutional neural networks. Scientific reports, 11(1), 1-11.

[29] Shwartz-Ziv, R., & Armon, A. (2022). Tabular data: Deep learning is not all you need. Information Fusion, 81, 84-90.

[30] Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., & Chen, Z. (2021). SySeVR: A framework for using deep learning to detect software vulnerabilities. IEEE Transactions on Dependable and Secure Computing, early access article.

[31] LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), 436-444.

[32] Sathya, R., & Abraham, A. (2013). Comparison of supervised and unsupervised learning algorithms for pattern classification. International Journal of Advanced Research in Artificial Intelligence, 2(2), 34-38.

[33] Lee, K., Booth, D., & Alam, P. (2005). A comparison of supervised and unsupervised neural networks in predicting bankruptcy of Korean firms. Expert Systems with Applications, 29(1), 1-16.

[34] Zhao, Zilong, Sophie Cerf, Robert Birke, Bogdan Robu, Sara Bouchenak, Sonia Ben Mokhtar, and Lydia Y. Chen. "Robust anomaly detection on unreliable data." In 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 630-637. IEEE, 2019.

[35] Robles-Velasco, A., Cortés, P., Muñuzuri, J., & Onieva, L. (2020). Prediction of pipe failures in water supply networks using logistic regression and support vector classification. Reliability Engineering & System Safety, 196, 106754.

[36] do Nascimento, P. P., Pereira, P., Mialaret, J. M., Ferreira, I., & Maciel, P. (2021). A methodology for selecting hardware performance counters for supporting non-intrusive diagnostic of flood DDoS attacks on web servers. Computers & Security, 110, 102434.

[37] Domenico Cotroneo, Roberto Natella, and Stefano Rosiello. 2017. A fault correlation approach to detect performance anomalies in Virtual Network Function chains. In Software Reliability Engineering (ISSRE), 2017 IEEE 28th Int. Symposium on. IEEE, 90–100.

[38] Cruz, T., Barrigas, J., Proença, J., Graziano, A., Panzieri, S., Lev, L., & Simões, P. (2015, May). Improving network security monitoring for industrial control systems. In 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM) (pp. 878-881) IEEE.

[39] Al, S., & Dener, M. (2021). STL-HDL: A new hybrid network intrusion detection system for imbalanced dataset on big data environment. Computers & Security, 110, 102435.

[40] He, P., Zhu, J., He, S., Li, J., & Lyu, M. R. (2017). Towards automated log parsing for large-scale log data analysis. IEEE Transactions on Dependable and Secure Computing, 15(6), 931-944.

[41] Chou, D., & Jiang, M. (2021). A survey on data-driven network intrusion detection. ACM Computing Surveys (CSUR), 54(9), 1-36.

[42] Zoppi, T., Ceccarelli, A., Puccetti, T., & Bondavalli, A. (2023). Which Algorithm can Detect Unknown Attacks? Comparison of Supervised, Unsupervised and Meta-Learning Algorithms for Intrusion Detection. Computers & Security, 103107.

[43] Boughorbel, Sabri, Fethi Jarray, and Mohammed El-Anbari. "Optimal classifier for imbalanced data using Matthews Correlation Coefficient metric." PloS one 12.6 (2017): e0177678.

[44] G. O. Campos, A. Zimek, J. Sander, R. J. Campello, B. Micenko-va, E. Schubert, I. Assent, and M. E. Houle, “On the evaluation of outlier detection: Measures, datasets, and an empirical study”, in Lernen, Wissen, Daten, Analysen 2016. CEUR workshop proceedings, 2016.

[45] Sater, R. A., & Hamza, A. B. (2021). A federated learning approach to anomaly detection in smart buildings. ACM Transactions on Internet of Things, 2(4), 1-23.

[46] Karim, F., Majumdar, S., Darabi, H., & Chen, S. (2017). LSTM fully convolutional networks for time series classification. IEEE access, 6, 1662-1669.

[47] Baydogan, M. G., Runger, G., & Tuv, E. (2013). A bag-of-features framework to classify time series. IEEE transactions on pattern analysis and machine intelligence, 35(11), 2796-2802.

[48] Ji, C., Du, M., Hu, Y., Liu, S., Pan, L., & Zheng, X. (2022). Time series classification based on temporal features. Applied Soft Computing, 128, 109494.

[49] Wang, T., Liu, Z., Zhang, T., Hussain, S. F., Waqas, M., & Li, Y. (2022). Adaptive feature fusion for time series classification. Knowledge-Based Systems, 243, 108459.

[50] Abanda, A., Mori, U., & Lozano, J. A. (2019). A review on distance based time series classification. Data Mining and Knowledge Discovery, 33(2), 378-412.

[51] Shi, W., Cao, J., Zhang, Q., Li, Y., & Xu, L. (2016). Edge computing: Vision and challenges. IEEE internet of things journal, 3(5), 637-646.

[52] Repository of datasets and experimental results (online), https://drive.google.com/file/d/1E7PYMnUKVqLTEzPkIKtxXidjDXdymTm9/view?usp=sharing

[53] Bruneo, D., Distefano, S., Longo, F., Merlino, G., & Puliafito, A. (2018). I/Ocloud: adding an IoT dimension to Cloud infrastructures. Computer, vol. 51, no. 1, pp. 57-65, January 2018, doi: 10.1109/MC.2018.1151016.

[54] Lin, J., Yu, W., Zhang, N., Yang, X., Zhang, H., & Zhao, W. (2017). A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications. IEEE Internet of Things Journal, 4(5), 1125–1142. https://doi.org/10.1109/JIOT.2017.2683200

[55] Yi, S., Hao, Z., Qin, Z., & Li, Q. (2020). Edge Cloud Computing in the Internet of Things. IEEE Internet of Things Journal, 7(5), 3969–3982. https://doi.org/10.1109/JIOT.2020.2967584

[56] Naha, R. K., Garg, S., Georgakopoulos, D., Jayaraman, P. P., Gao, L., Xiang, Y., & Ranjan, R. (2019). Fog Computing: Survey of Trends, Architectures, Requirements, and Research Directions. IEEE Access, 7, 99983–10009. https://doi.org/10.1109/ACCESS.2019.2936871

[57] Belkhir, L., & Elmeligi, A. (2020). Assessing ICT global emissions footprint: Trends to 2040 & recommendations. Journal of Cleaner Production, 177, 448–463. https://doi.org/10.1016/j.jclepro.2017.12.239

[58] Mahmud, R., Kotagiri, R., & Buyya, R. (2018). Fog Computing: A Taxonomy, Survey and Future Directions. In Internet of Everything (pp. 103–130). Springer, Singapore. https://doi.org/10.1007/978-981-10-5861-5_5

[59] Puliafito, C., Mingozzi, E., Vallati, C., Longo, F., & Merlino, G. (2018). Virtualization and Migration at the Network Edge: An Overview. In 2018 IEEE International Conference on Smart Computing (SMARTCOMP), Taormina, Italy, 2018, pp. 368-374, doi: 10.1109/SMARTCOMP.2018.00031.

[60] Bock, C., Aubet, F. X., Gasthaus, J., Kan, A., Chen, M., & Callot, L. (2022). Online time series anomaly detection with state space gaussian processes. arXiv preprint arXiv:2201.06763.